Robust concurreny

Implementing basic concurrency patterns.

GOLAB 2019, 2019–10–22, 14:30-16:00, Florence

Martin Czygan

About me

SWE @ubleipzig.

- Leipzig University Library is involved in a variety of open source projects in the library domain: catalogs, repositories, digitization and image interop frameworks (IIIF), data acquisition, processing and indexing tools
- Go for tools and services
- Co-organizer of Leipzig Gophers

Background

Moderate use of classical concurrency tools, like threads (in Java) or multiprocessing. Use case: implementing data and legacy system access tools. Code is slow (for many reasons). Investigating parallelization approaches. Discover ZeroMQ, which looks like an embeddable networking library but acts like a concurrency framework.

Patterns, resulted in faster tools. Discovered sequential, than concurrent Go. Better resource utilization. Small parallel tools.

Goal

Get more familiar with a few primitives and patterns.

- Slides
- Example code
- A few exercises
- Pop quizzes with questions related to a paper

Paper: Understanding Real-World Concurrency Bugs in Go

- Looked at Docker, k8s, etc, CockroachDB, gRPC-Go, BoltDB

> We made nine high-level key observations of Go concurrency bug causes, fixes, and detection.

Axes:

- Classic vs CSP style primitives
- Blocking vs Nonblocking bugs

If not noted otherwise, "Paper" will refer to this paper.

A few little projects

- A parallel link checker.
- A generic parallel line processing function.
- Fan-out indexing with solrbulk and esbulk.
- Hedged request with networked version of the fortune command.

Concurrency is hard

It's so hard, you may want to avoid it completely.

- Example: x/hard

Advice from https://golang.org/ref/mem

> Don't be clever.

Example: x/hard

Question: What do you think can happen?

Example: x/hard

- nothing is printed
- value = 0
- value = 1

Example: x/hard

> Most of the time, **data races are introduced because the developers are thinking about the problem sequentially**. They assume that because a line of code
> falls before another that it will run first. They assume the goroutine above
> will be scheduled and execute before the data variable is read in the if statement. (CIG)

Go memory model

> Many compilers (at compile time) and CPU processors (at run time) often make some optimizations by **adjusting the instruction orders**, so that the instruction execution orders **may differ from the orders presented in code**. **Instruction ordering** is also often called **memory ordering** (GO101)

Go memory model

Happens-before.

> Within a single goroutine, reads and writes must behave as if they executed in the order specified by the program. That is, compilers and processors may reorder the reads and writes executed within a single goroutine only when the reordering does not change the behavior within that goroutine as defined by the language specification.

Within a single goroutine, the happens-before order is the order expressed by the program.

Go memory model

> Within a single goroutine, there is no concurrency, so the two definitions are equivalent: a read r observes the value written by the most recent write w to v. When multiple goroutines access a shared variable v, they **must use synchronization events to establish happens-before conditions that ensure reads observe the desired writes**.

Example: x/hardsleep

```
        // Note: Extensive testing found 5µs to be the ideal time to have chance to
        // observe different results on subsequent runs. Works on my machine.
        time.Sleep(5 * time.Microsecond)
```

Example: x/hardsleep

> The takeaway here is that you should always target logical correctness. Introducing sleeps into your code can be a handy way to debug concurrent programs, but they are not a solution. (KCB)

Pop quiz

The paper looks at goroutine creation sites (go keyword). Named and anonymous functions can be used.

Question: What is more frequent?

- A: named functions
- B: anonymous functions
- C: depends on the project

Pop quiz

Question: What is more frequent?

- B: anonymous functions
- C: depends on the project

Only BoltDB is different, otherwise anonymous functions seem to be more popular.

Go concurrency primitives

Go support classic and CSP style. Which one do you choose? (KCB)

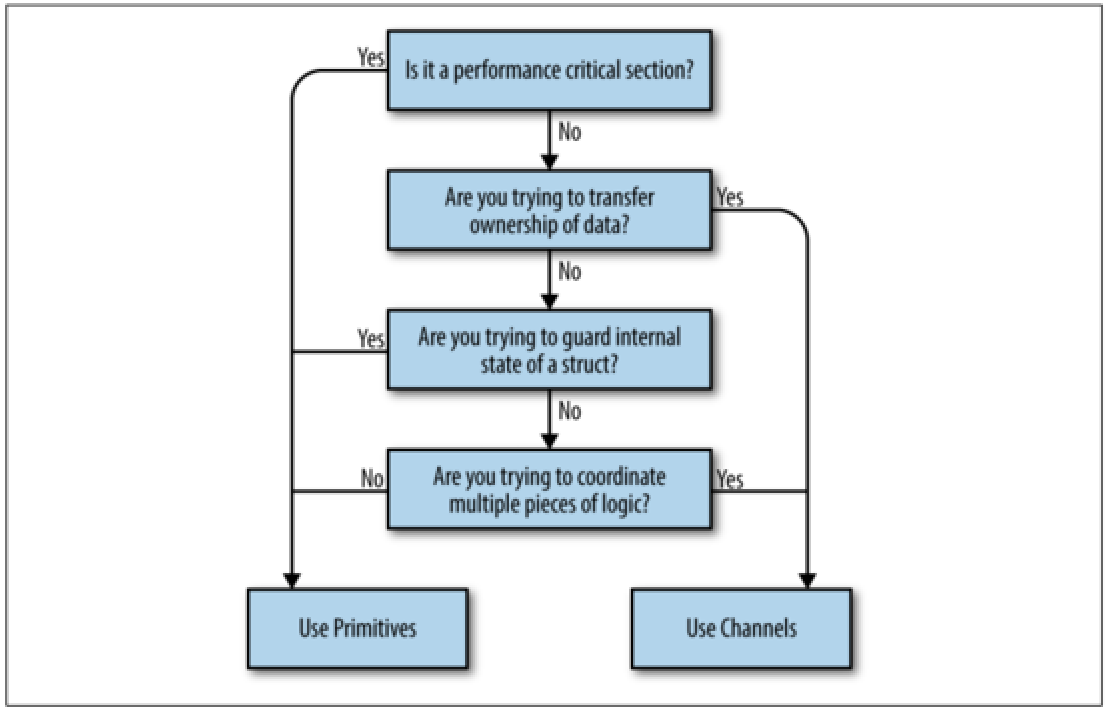

Figure 2-1. Decision tree

Pop quiz

The gRPC project has several implementations, e.g. C or Go. The Paper discribes the ratio of **normal source lines** and **goroutine / thread creation sites** as goroutine / thread creations per 1000 lines of code. Example: 1 go statement per 1000 lines would be: 1.

Question 1: What do you think this ratio is for the gRPC-Go project?

- A: 0.083
- B: 0.83
- C: 8.3

Question 2: Is the ratio for gRPC-C higher or lower?

- A: higher
- B: lower

Pop quiz

Answer: B, B.

- gRPC-Go: 0.83, gRPC-C: 0.03

It seems that people use the facilities available.

Classic Data Race

Informal definition: A data race occurs, if multiple threads access a shared resource (variable) and at least one of the accesses is write.

Exercise

Edit: x/counter/racy.go

```go
// Exercise: Update a variable from different goroutines.
//
// (1) Start 100 goroutines.
// (2) Each goroutine should increment the counter variable (c).
// (3) Before main exits, print the value of c.

func main() {
	var c int // Count this up from different goroutines.
}
```

Exercise

The printed result will be inconsistent (e.g. 93, 96, 100, ...).

The race detector

The Go has a builtin race detector. It can detect these kind of errors.

- https://golang.org/doc/articles/race_detector.html

> Data races are among the most common and hardest to debug types of bugs in concurrent systems. A data race occurs when two goroutines access the same variable concurrently and at least one of the accesses is a write.

Example

```
[x/counter] $ go run -race racy.go
$ go run -race racy0.go
WARNING: DATA RACE
Read at 0x00c0000b4010 by goroutine 8:
  main.main.func1()
      /home/tir/code/miku/concgo/solutions/counter/racy0.go:12 +0x38

Previous write at 0x00c0000b4010 by goroutine 7:
  main.main.func1()
      /home/tir/code/miku/concgo/solutions/counter/racy0.go:12 +0x4e

Goroutine 8 (running) created at:
  main.main()
      /home/tir/code/miku/concgo/solutions/counter/racy0.go:11 +0x83

Goroutine 7 (finished) created at:
  main.main()
      /home/tir/code/miku/concgo/solutions/counter/racy0.go:11 +0x83
```

Race detector

The race detector uses another library called thread sanitizer (TSan) and works by inserting extra statements for tracking into the executable. The executable gets slower and will consume more memory.

Race detector limits

> The **race detector only finds races that happen at runtime**, so it can't find races in code paths that are not executed. If your tests have incomplete coverage, you may find more races by running a binary built with -race under a realistic workload.

But it found numerous bugs in the standard library and elsewhere.

Fixing the counter

Use a `sync.Mutex`, which serializes access to a shared variable, it has two methods:

- Lock
- Unlock

> We can define a block of code to be executed in mutual exclusion by surrounding it with a call to Lock and Unlock as shown on the Inc method.

We can also use `defer` to ensure the mutex.

Exercise: x/counter

Edit: x/counter/save.go

```
// Exercise: Write a save counter.
//
// (1) Create a type counter that wraps an int and uses a sync.Mutex to protect access to the value.
// (2) Create a method on the type named `Inc` that increment the counter by one.
// (3) In main, create a counter, and start 100 goroutines, each incrementing the counter.
// (4) Print out the final value of the counter (by just accessing the struct field).
//
```

Counter wrap up

- sync.Mutex for protecting access to a resource
- A mutex will have performance implications, but these may matter later
- A sync.RWMutex allows multiple reads, but only a single write
- You can embed a struct to make the API a bit simpler
- For counters, there are alternatives, like `atomic.AddUint64`

Worker pools